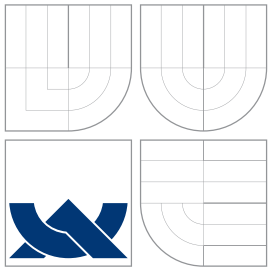
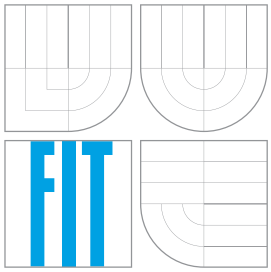


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

HLEDÁNÍ REGULÁRNÍCH VÝRAZŮ S VYUŽITÍM TECHNOLOGIE FPGA

FAST REGULAR EXPRESSION MATCHING USING FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN KAŠTIL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KOŘENEK
SUPERVISOR

BRNO 2008

Abstrakt

V práci je vysvětluje několik algoritmů pro vyhledávání výrazů v textu. Algoritmy pracují v software i hardware. Část práce se zabývá rozšířením konečných automatů. Další část práce vysvětluje, jak funguje hash a představuje koncept perfektního hashování a CRC. Součástí práce je návrh možné struktury vyhledávací jednotky založené na deterministických konečných automatech v FPGA. V rámci práce byly provedeny exprimenty pro zjištění podoby výsledných konečných automatů.

Klíčová slova

IDS, konečné automaty, CRC, perfektní hashování, vyhledávání vzorů, programovatelný hardware, FPGA

Abstract

The thesis explains several algorithms for pattern matching. Algorithms work in both software and hardware. A part of the thesis is dedicated to extensions of finite automatons. The second part explains hashing and introduces concept of perfect hashing and CRC. The thesis also includes a suggestion of possible structure of a pattern matching unit based on deterministic finite automatons in FPGA. Experiments for determining the structure and size of resulting automatons were done in this thesis

Keywords

IDS, finite automaton,CRC, perfect hashing, pattern matching, programmable hardware,FPGA

Citace

Jan Kaštil: Fast Regular Expression Matching Using FPGA, diplomová práce, Brno, FIT VUT v Brně, 2008

Fast Regular Expression Matching Using FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Kořenka

. . . . . .
Jan Kaštil
May 18, 2008

Contents

Chapter 1

Introduction

The development in the field of computer networks is fast. Today, theoretical performance is up to 100Gb/s networks. Fortunately, currently implemented networks are not faster than 10 - 40 Gb/s.

This fast development allows internet connection to become publicly available. This availability brings many malicious users, who create potentially harmful traffic. In the past a simple firewall was enough to filter out malicious traffic. Each protocol had its specific port number and therefore the protocol could be simply identified by looking at this number. Due to easy structures of the networks and a small number of the attacks this protection was satisfactory in many positions. Only the specified protocols could go through the firewall and attack on these protocols were rare and therefore could be solved on each server individually.

Today the number of attacks rapidly grows. In is not uncommon, that some big networks are under a constant attack. Malicious users are trying to bring their computers down and use them for their own purposes. They are using more sophisticated techniques. Malicious traffic masks itself as normal data transitions. Many modern protocols operate without a predefined port number. And some even use a port which is allocated for different protocols. Therefore the firewall is not able to stop malicious traffic only by looking at the port number. Even if we were able to identify the protocol by some other means, it still would not solve the problem, because too many attacks would go through such a defense.

It is not always possible in modern networks to have all servers secured to all forms of attack. The problem often is that the server is not operated by skilled administrators. Therefore all malicious traffic should be stopped at the entering point of the network. This is done by Intrusion Detection System (IDS). One of the most spread IDS is Snort [26]. Snort is a signature based IDS. This means that Snort scans content of traffic for predefined patterns. The pattern could be a string or even a regular expression. The number of patterns in Snort's database is increasing. Snort is software and therefore it is limited by power of the CPU and the bus speed. Even in the currently fastest PC snort obtains throughput only in order of hundreds Mb per second. This indicates that there is a great need for some hardware acceleration. The 80% of all Snort's work consists of pattern searching. And this is a motivation for developing hardware acceleration for pattern matching.

Several solutions use ASIC to achieve throughput high enough for modern networks. But these systems cost much more than software solutions and the function flexibility is always limited. Therefore many methods used Field Programmable Gate Array (FPGA) to implement a hardware accelerator. Using FPGA is much cheaper than ASIC and there is

the possibility to change the function of hardware only by downloading new firmware into the device.

The thesis is divided into a few sections. In the theoretical section the basics of current methods are described without any progressive optimizations. After describing these methods, HW implementation of this problem is briefly described. But even with hardware acceleration, current methods are not optimal for widespread use. Third subsection of theory shows several possibilities of increasing throughput of finite automatons. The last subsection of theoretical introduction is presents the basics of hashing. The second section of this work is dedicated to the description of the newly proposed solution of this problem. Implementation of the library for work with finite automatons forms a part of this work. The purpose of this library was to allow testing of theoretical results and therefore the library was not optimized for fast execution.

Chapter 2

Theoretical background

This chapter introduces theoretically the problem of fast pattern matching. The initial part describes the software approach for searching string patterns. Some of the described algorithms can be extended for regular expressions. The second part describes the hardware approach to the same problem. Even with hardware acceleration, basic methods are not fast enough. Therefore the third section presents several optimizations for Finite automatons. The last part of theory introduction deals with hashing.

For the purpose of the next two sections, we can define a pattern as follows. Let us have text string p of length m. String p is called a pattern. Then let us have text string s of length n. Pattern searching is a process of finding positions of all substrings of s which are equal to the pattern p. Let $s[i]$ be the i-th symbol of string s.

2.1 Searching algorithms

This section describes a few basic searching algorithms. Namely it is Bruteforce [8] algorithm, Boyer-Moore [25], Finite Automatons and their extension known as AhoCorasic's automaton [1]. These algorithms are developed for computers with a single processor which works in serial fashion.

Brute force algorithm

The algorithm works from the left to the right and goes through the whole string s. It also goes through all symbols in pattern p. Let us suppose that number i is a position in the input string and number j is a position in pattern p. Both numbers are initialized to 0 and symbols $p[j]$ and $s[i+j]$ are compared. If the symbols are equal, number j is increased by one and the comparison is repeated. If not, number j is set to 0 and number i is increased by one.

When number j is equal to the length of the pattern, then i points to the first character of the pattern in the input string. If the search is done only for the first occurrence of the pattern, then the algorithm ends. For all occurrences number i is saved into a list and the algorithm continues by resetting number j and increasing number i. Searching ends if number i plus the length of the pattern is greater than the length of the input string.

This algorithm is easy to implement and understand. But Brute force algorithm is a very ineffective, because it needs almost $m * n$ character comparisons.

Boyer-Moore Algorithm

Boyer-Moore requires two steps. The first step is done without knowledge of the input string and therefore it is called preprocessing. Two tables are computed during preprocessing: the first table has the size of input alphabet and the second table has the size of pattern.

The searching itself is done from the right to the left. If a mismatch occurs, which is probable in common language, algorithm uses the first table and finds the precomputed number for this character. There are two possibilities. First, the character does not exist even in the pattern. In this case the algorithm moves forward skipping a group of characters of the same length as the length of the pattern. Another possibility is that the actual character is contained in the pattern but in a different position. In such case the algorithm jumps forward over such number of characters, that character in actual position in input string match character in pattern. These steps both use just the first table but they work fine only if the mismatch occurs with the first tested character. The second table has to be used if the mismatch occurs after several matched characters. The second table represents our knowledge about the pattern. The basic idea behind the second table is that if the suffix of the pattern is contained in the pattern in some other position algorithm can jump forward over such number of characters that all previously tested characters match again. This saves several comparisons. The lower number from both tables is taken as the length of the jump.

Boyer-Moore algorithm is considered to be one of the most efficient algorithms in usual applications and therefore it is widely implemented. Time complexity of this algorithm could be $O(m/n)$. The worst case is $O(3m)$ for searching for the first occurrence and $O(mn)$ for searching for all occurrences. Another problem with this algorithm is that it is not suitable for searching for several patterns at once. Searching for each pattern has to be done sequentially.

This algorithm works faster on longer patterns, because longer patterns can be moved much farther than shorter ones. But this algorithm is not sufficient for small alphabets, because in small alphabets, characters are frequently used and thus the first table will not bring any performance increase.

Finite Automaton

Next approach to pattern searching is Finite Automata (FA). Nondeterministic Finite Automata (NFA) [31] is quintuple $M = (Q, \Sigma, \delta, q_0, F)$
where: Q is finite set of states
Σ is finite input alphabet
δ is transition function: $Q \times \{\Sigma \cup \epsilon\} \rightarrow 2^Q$
q_0 is start state
$F \subset Q$ such that F is set of end states

A NFA decides whether an input string belongs to the language defined by this NFA. In each step, the NFA reads one character from the input string and performs a transition. When the whole string is read and the NFA has active some of its end states, then the answer is yes and the string belongs to the language of this automaton. In other cases, the string does not belong to the automaton's language. But finite automata can be used for searching as well. If a NFA enters its finite state, it means that it has just read a string which belongs to its language, in other words, a string which was searched for. Therefore if a NFA is used for searching, it has to remember the position, when it entered its finite state and return it as the last character of the searched pattern.

Transition can be performed, if function δ is defined for the active state and the first char of the input string or if δ is defined for the active state and character ϵ. If δ is defined for ϵ then no character is read from the input string. A transition is performed by changing the active state. The result of the transition is a change of the active state. The active state always becomes inactive and a state resulting from the transition is activated. Choosing an active state from the returned set is nondeterministic, in other words, the new active state is chosen randomly. A string is accepted by the NFA if there exists any way from the starting state to one of its end states and this way can be performed by reading all characters from the input string. The right active state has to be chosen in each step of the automaton. This means that all possible combinations must be tested or there is the need for an oracle function which returns the right state.

This form of automaton is well suited for connection of several automatons in serial or parallel ways. Serial connection is a simple concatenation and parallel connection means that the language of the new automaton is a join of languages of the respective two automatons. But ϵ transition takes the same execution time as other transitions and does not read any input character. Therefore there is a need for definition of a NFA of a new version, which does not have ϵ transitions. In that case δ function is defined like this: $Q \times \Sigma \rightarrow 2^Q$. This new version of NFA can be easily created from previously described NFA [13].

If such NFA uses previously mentioned oracle function, its time complexity is O(n). Such oracle function can be implemented by a little trick. There could be only one active state in the previously mentioned NFAs. By allowing each state to be active all transition could be done by NFA and therefore if there is a way leading from the start state to the end state, then the end state becomes active. Execution time of such automaton is also increased. Its time complexity is $O(na)$ where $a = |Q|$.

It is possible to reduce the time complexity of such automaton back to $O(n)$. Each transition leads to a set of states and all states in this set have to be active. The reduction is achieved by defining a new state, which will be equivalent to the whole set. Function δ can return only this state and if all transitions are changed there will always be only one active state. The problem connected with this approach is an increase in the number of states and therefore an increase in the size of the transition table.

Deterministic Finite Automaton (DFA) can be defined as follows. Deterministic Finite Automata [31] is quintuple $M = (Q, \Sigma, \delta, q_0, F)$
where: Q is finite set of states
Σ is finite input alphabet
δ is transition function: $Q \times \Sigma \rightarrow Q$
q_0 is start state
$F \subset Q$ such that F is set of end states

The only difference between a NFA and a DFA is that the DFA function δ always returns only one state and therefore there is always only one possible transition. This enables the DFA to accept a string in i$O(n)$ time complexity. The DFA can be obtained from a NFA [13]. The number of states in a DFA is much bigger than the number of states in a NFA, therefore there is the need to create Minimal DFA. A minimal DFA is the smallest DFA accepting the specified language and there is always only one Minimal DFA for one language with the exception of isomorphism. A description of the algorithm to convert a DFA into its minimal form can be found in [13].

In the previous text the transitions were assumed to take a constant time without any discussion whether it is even possible. Time complexity of transitions depends on time complexity of δ function. Also memory consumption of a DFA implementation depends on

the implementation of function δ.

Function δ can be implemented as a table and then its evaluation takes a constant time but the table requires $|Q||\Sigma|$ memory, which is unacceptable for bigger automatons.

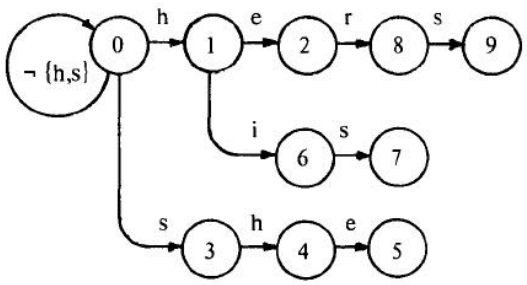
Aho-Corasick

Aho-Corasick is a simple and efficient algorithm for locating all occurrences of any of the finite number of keywords in an input text. Aho-Corasick consists of two steps. The first step is construction of a finite state pattern matching machine and the second step is using this machine to search through the input text. The pattern matching unit consists of a set of states and three functions. Function called Goto is an equivalent of the transition function from Finite automatons. Goto leads from one state into another only if a specified character was read. Goto function is sometimes represented as a graph as in figure 2.1. The second function is called failed. This function assigns each state of the pattern matching unit its successor. Function fail can be done only if function goto can not be used. The last function is output function. Output function assigns states to its keywords. If a keyword is found, function output returns which keyword was found before the search for other keywords continues.

This algorithm is able to search for a whole set of patterns by one pass through the string. In the worst case time complexity of such state machine will be $O(n)$. In fact Aho-Corasick will never need more than $2n$ steps, where n is length of the input string.

For details on how to create this Finite State Machine read [1]

Fig. 1. Pattern matching machine.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Figure 2.1: Representation for Aho-Corasick searching for keyword : he, she, his, hers from [1]

2.2 Hardware accelerations of pattern searching

Several methods of pattern matching in hardware is described in [28] and [16]. The following text deals with the most interesting methods used in hardware.

Knuth-Moris-Prat Algorithm

Hardware acceleration of KMP algorithm uses multiple units, one unit per pattern. Many patterns must be searched in IDS. A chain of KMP units can be implemented. The great advantage of this methodology is the possibility of changing the patterns without reconfiguration of FPGA. This is because KMP algorithm uses a table to represent the structure of a pattern. This table is precomputed in PC and uploaded into the memory of a KMP unit. This upload is a very fast procedure compared with reconfiguration of FPGA. The disadvantage of KMP in hardware is that KMP is not able to read more than one character from the input string. Sometimes KMP even needs more than one comparison per character. Several comparisons per character can be solved by implementing two comparators per unit. But even then the throughput of KMP is still limited by accepting only one character per cycle.

Content Adresable Memory (CAM)

In this architecture, patterns are stored in CAM and input data is used as key to the memory. Data in CAM has to be inserted in all possible shifts, which means if length of the pattern is limited to 15 characters then 15 possible shifts of the pattern has to be stored in CAM. The advantage of this method is that whole searching is done simply by looking into the memory and therefore changing the pattern set requires only uploading a new set of pattern into CAM. Unfortunately this method is not able to search for regular expressions.

Finite Automata

NFA in hardware is potentially advantageous over software implementation. NFA in software has time complexity $O(na)$ where $a = |Q|$. Hardware implementation can have processing one processing unit per state and therefore it is possible to have time complexity $O(n)$ at the cost of a processing units. Processing units do not need to be very complex. In fact, processing units for NFA are implemented as one bit registers and a logical function for setting this register. The logical function can be implemented simply by oring all input transitions, where input transitions are represented as logical function AND of the value of the register from which the transition goes and the input symbol. This schema allows using a shared decoder. The decoder takes at its input the characters from the input string and the output is one bit value which is used as the input to the logical function realizing the transition.

This method does not need any memory in FPGA chip but whole NFA is implemented in logical cells of FPGA. This can be considered both a disadvantageous and at the same time advantageous. It is a disadvantage because if there is a need for a change of the set of patterns then representation of NFA has to be synthesized from its VHDL representation all over again and then FPGA must be reconfigure. Both of these steps take a long time and therefore this approach is not suitable for systems which need fast changes of the pattern set. Memory on FPGA is in fact limited; therefore the fact that NFA implementation does not need it can be considered a great advantage.

Deterministic Finite Automaton always has only one processing unit. Its structure is stored in memory and therefore it is easy and fast to download different rule. Processing units of DFA have to be more complicated than processing units of NFA. It has to have k bit register to store the number of active state and a block which is called next state logic. Next State logic block is used to compute the number of state which will be active after the transition from active state and the first symbol of the input string. In fact, Next state logic is hardware implementation of function δ. As mentioned before, it is problematic to store the transition table uncompressed in software and it is impossible in the limited memory of FPGA. Therefore quality of implementation of Next State Logic determines quality of the whole solution.

Bloom filters

A Bloom filter is a structure which is able to tell if a string belongs into a set. The biggest advantage of a Bloom filter over other solutions is that a bloom filter always recognizes a string from the set. A Bloom filter can have false positives but never false negatives. Probability of false positives is determined by the memory size used for storing a Bloom filter.

A Bloom filter is realized by several hash functions and one bit field. The filter contains k hash function. The input string is hashed by all functions. Results of each function are then used as index into the bit field. If the Bloom filter is taught a new string, the bit at this index is set to 1. If the Bloom filter is used to tell if a string belongs to its set, then the value in the bit field is compared to 1. If for all hash functions the values on their positions are 1, then the string is considered to be in the set. The size of the bit field determines probability of false positives. This probability can be reduced by increasing number k.

Hashing in pattern searching

Details of hashing is discussed in section 2.4. The basic idea of this approach is that the input traffic does not need to be tested to all possible patterns but only to one which is in the packet. Therefore authors [27] introduced the idea of a hash tree which for each packet returns one possible pattern. The pattern is subsequently downloaded from memory into a comparator and the packet is tested only for one pattern. The disadvantage of this approach is that their hash tree is implemented in FPGA logic. Therefore, if the pattern set needs to be changed, users have to reconfigure FPGA and synthesize a new hash tree which is a time consuming operation.

Comparisons of approaches

For more detailed information about this comparison look at [16] or [27]. The first column of the table describes the method. Second column contains the type of device in which the method was implemented. In the third column is the number of bits which is accepted by one cycle. And the last column shows the throughput in Gb per second. It can be seen from the table that NFA approach offers the best results. This is because Finite Automatons accept almost arbitrary number of character per cycle.

Descriptions	Device	Input bit per cycle	Throughput Gb/s
KMP [4]	Virtex2Pro-4	8	1,8
NFA [9]	Virtex2P-125	128	16,516
Optimized CAM [3]	Virtex2P-100	64	8,840
Bloom filters [2]	VirtexE-2000	8	0,502
Hashing [27]	Virtex2-1500	16	5,734

2.3 Optimization of Deterministic Finite Automaton

A searching unit implemented using DFA has time complexity $O(n)$. This means that it needs constant time for each character in the input string and that each character in the input string is read only once. Frequency of FPGA depends on the design but commonly it is about 100 MHz. Therefore Classical DFA has throughput around 800Mb/s, which is too slow for modern networks. Therefore advanced optimizations of DFA approach is needed to meet requirements of modern networks.

2.3.1 Multichar Finite Automaton

Usually we use data type char as an alphabet for our automatons. This is because 8 bit datatype is commonly used in computer world and patterns which are searched for are written by humans who also use 8bit alphabet. Multichar Finite Automaton has a bigger alphabet than just 8bit and therefore it is able to accept more than one character per transition. Such automaton can be easily built from a normal automaton. NFA automaton for word HALLO is in figure 2.2 and the same automaton extended to accept two character per cycle is in figure 2.3. Symbol * stands for every possible character and is called DNC (Do Not Care).

Because such automaton accepts k character per cycle, throughput is increased k times at the cost of an increase in the transition table.

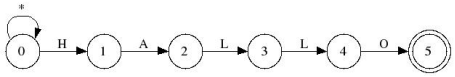


Figure 2.2: NFA searching for word HALLO

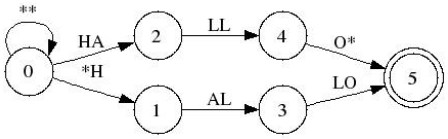


Figure 2.3: NFA searching for HALLO accepting 2 chars per step

A multichar automaton will have the same number of states but a different number of transitions. Therefore, the new automaton is created without transitions but with the same states as previous automaton. Transitions of the new automaton are specified by the start state, end state and succession of characters which is accepted by this transition. Extension of the automaton is done in three steps.

1. Creation of transitions from Start State
   In this step all transitions which lead from the start state are computed. Transitions leading from the start state accept any number of characters per cycle up to the number to which the automat is extended. The end state for a specific transition is computed as closure of transitions of size n, where n is the number of characters accepted per cycle. Transitions which accept less than n characters are completed by DNC from the start.

2. Creation of transitions of specified length from normal states
   Transitions defined in this steps always accept precisely n characters. Transitions are computed as closure of size n over transitions of the source automaton. If some transitions should point out of the automaton, for example the transition from state 4 in 2.2, then such transition will not be created.

3. Creation of transitions which lead into the end state
   Transitions created in this step accept less than n characters. Therefore these transitions are completed by DNC added at the end of their description. Reverse closure of all sizes lesser than n are computed for each end state and a transition is added from each state in closure to its end state. The transition is added only if it accepts less than n characters. Figure 2.4 shows an example, when there could be infinite number of transitions created from state 1 to state 3 but only transitions which accept less than n characters are added into the transition table. Transition which accepts precisely n characters was created in the previous step.

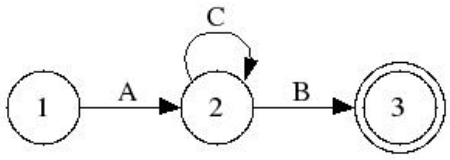


Figure 2.4: Situation which creates several successions of different length

The algorithm briefly described above looks fine, but fails with patterns like HALLO containing arbitrary number of O at the end. Figure 2.5 shows a normal automaton and figure 2.6 shows a multichar NFA created by the described algorithm. This automaton is not correct because it accepts strings such as HELLOaOOOO. This is because the algorithm excepts that the search ends when the pattern is found. The same problem can be seen with the start state. It lies in the fact that the algorithm creates cycles in the graph that contains DNC. This can be solved by adding one additional step into this algorithm.

Each end state which at the same time has an outgoing transition is divided into two end states. The target of the transition which contains DNC will be change from the original end state to its copy. This solves the issue of cycles with end states. Cycle in the start state is solved similarly. A new state is created and all transitions without DNC which lead from the start state are copied into the new state. All transitions which lead into the start state are subsequently redirected into this new state.

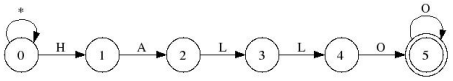


Figure 2.5: NFA for HELLO with arbitrary number of O

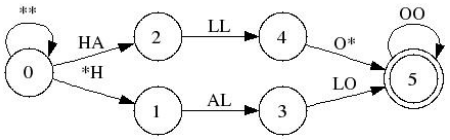


Figure 2.6: Wrong Multichar for HELLO with arbitrary number of O

2.3.2 Sets of patterns

Using DFA requires trade between memory consumption and speed of searching. There are many regular expressions in IDS, which are searched for. A naive approach implements one searching unit for each regular expression. This leads to problems of time complexity in software, because it has time complexity $O(cn)$, where c is the number of regular expressions. In a hardware implementation, it would be possible to have time complexity $O(n)$ at the cost of c parallel searching units. But if the number of searching units depends on the number of regular expressions, the advantage of DFA over NFA is lost because during the change of the pattern set, a reconfiguration is needed to adapt the number of searching units.

Another solution is merging all regular expressions into one. Resulting DFA needs too much memory even for software solutions and therefore implementation in FPGA is impossible. There are no exceptions of DFA the symbol tables of which have more than 1GiB in current software implementations.

This problem is addressed by Grouping patterns. The number of states of an automaton created by joining two smaller automaton could be almost $|Q_1| \times |Q_2|$, where Q_1 and Q_2 are sets of states of two smaller automatons. Two regular expressions for which such state explosion occurs can be called colliding regular expressions. Grouping pattern creates such groups of patterns ensuring that each group contains the minimal number of colliding patterns.

This idea together with method of its implementation was introduced in [30]. They introduce a method for finding such regular expressions in a given set so that this regular expression will have minimal collisions. A naive approach would be to compute all possible groups and then choose the best result. This approach would work, but it would take too much computation power. Therefore [30] comes with heuristics. In the first step, they compute how regular expressions interact with each other. The result of this step is a graph (V, H). V is a set of vertices and H is a set of edges. Each vertex from V corresponds to one regular expression from the given set. Minimal DFA is subsequently computed for each two regular expressions and for their merge. If the merged DFA is bigger than the sum of DFA for rules, then the edge between these two vertices is added to the graph.

The second step uses previously constructed graph to create groups of regular expressions. Each group has a limit on size of its Minimal DFA, but by merging two automatons, the result is always a NFA. Therefore after adding a new rule into the group, Minimal DFA

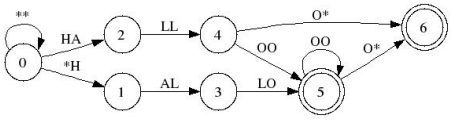


Figure 2.7: Correct Multichar for HELLO with arbitrary number of O

has to be computed again. Two situations can occur: first, the group is empty. In this case, the algorithm looks into the graph from the previous step and finds the vertex, which has minimal number of edges. Then it adds the rule corresponding with this vertex into the new group and deletes the vertex from the graph. In the second situation, the group already contains several rules. Now the algorithm chooses the vertex from the graph which has minimal number of edges with vertices which already belong to the group. After finding such vertex, its rule is added into the group and Minimal DFA is computed for the whole group. If Minimal DFA is smaller than the limit, chosen vertex is deleted from the graph. But if Minimal DFA of the group is bigger than the limit, the new rule is deleted from the group and the group is closed. This means that no other rule can be added into this group. Cycle continues until there is a rule which does not belong to any group.

2.3.3 Delayed DFA

This method was proposed by [18]. The purpose of Delayed DFA (DDFA) is to decrease the size of the transition table of DFA. This is done by introducing so called Delayed transitions. These transitions can be done only if no other transition is possible. The problem is that Delayed transitions do not consume any character from the input string and therefore it increases the time needed for searching. Time complexity is $O(mn)$, where n is the number of characters in the input string and m in length of the longest path from delayed transitions. According to authors, introducing delayed transitions into DFA can compress the transition table by more than 95%.

There are usually many strings in which searching is done. Each packet can be searched independently. In such case, lesser throughput is enough. DDFA is therefore acceptable for such IDS systems. These IDS have some disadvantages compared to IDS that use the whole traffic but discussion of such disadvantage is out of the topic of this work. Authors suggested using several searching units but shared memory for the transition table. This introduces another savings of memory but there is a possibility that two similar packets will block each other, because both automatons will try to read different parts of the same memory. Therefore several DDFA must be equipped with an algorithm allowing intelligent distribution of the transition table among several memory cells and limit the probability that two or more DDFA will try to read from the same cell.

2.4 Hashing

As mentioned in the previous sections, hashing can be a very useful tool in IDS. Therefore the whole section is dedicated to hashing. Hashing is used in many parts of computer science. This section briefly introduces the most important uses of hashing function and then it concentrates on the areas of hash function which could be used in IDS, mainly perfect hashing.

2.4.1 Introduction to hashing

For more advanced introduction the reader can use for example [20]. Hash function is a mathematical function, which produces deterministic output with the same bit length for each input. Almost every function has this property and therefore almost every function can be used as a hash function. But not every function will be a good hash function. The quality of a hash function can be assessed according to the fulfillment of these requirements:

- Hash function must be fast
- Hash function must have uniform distribution in output

There are many other requirements for good hash functions but they depend on application of these hash functions.

Formal Definitions

Lets have universum U and interval of integers $M =< 0, m-1 >$. Then function h can be defined as h: $U \rightarrow M$. Let S be a set which contains elements from U, which is used as inputs into hash functions. This set is not used in all applications of hash function because sometimes S can contain whole U or S is not known. Because $|U| \gg m$ there has to be such $u_1 \in Uu_2 \in U$, that $h(u_1) \wedge h(u_2)$. Then u_1 and u_2 are called synonyms. And the situation, when $u_1 \in S \wedge u_2 \in S$ is called collision.

Hash function in error detection

In error detection there is the need for a function that will change the output when errors in transferred data occur. There are two common types of errors during communication. Firstly, there is a possibility that a bit is changed independently on others. Secondly, there exists the so called burst error. This means that from the start of the burst error to the end of it all bits are wrong. Therefore, hash function used in error detection is chosen to allow detection of long burst errors and to change output for big Hamming Distance between the correct and mistaken message rather than other requirements or properties.

Hash tables

The array which stores data can be seen as a table, while hash function is used for indexing it. Collisions can be a big problem in hash tables because two records can not be stored at the same position in the array. Normal hash function always has collisions. There are many approaches to solving this problem.

- Hash function returns index into an array of lists and data is stored in the list. The problem of this approach is that in the worst case every key is mapped into the same list and therefore searching in such table becomes searching in that list and it has linear complexity.
- Data is stored in free positions in the hash table. If hash points at the position which already contains a value, index into the hash table is increased by some value and used as a new index. This is repeated until a free position is found. The problem of this method is clustering. This means that in the array there are clusters of positions which are full and then clusters of free positions. Clusters cause worse searching time. Many

modifications were introduced to solve this problem. Namely it is Double hashing, in which we use two hash functions and the result of the first hash function is increased by result of the second one or Quadratic hashing in which the index into the table is increased by a quadratic function.

- Cuckoo hashing is a method that makes sure that in the worst case two hash functions are computed. Using cuckoo hashing requires knowledge of set S. Basically, we have functions h_1 and h_2. Data are added at the position given by h_1 if some data were already there, then they are removed. Hence the name cuckoo hashing. Removed data are hashed by function h_2. If the returned position is free then the hashing ends but if there is another value, then all the steps are repeated. Repetition ends if a free position is found. With such ending condition it is possible to enter to an infinite loop. In such case the algorithm generates a new couple of hash functions and rehashes the whole table. This approach was first introduced by [24] and it is supposed to be a replacement for perfect hashing in some applications.

Hash function for hash tables must be quickly computed and must uniformly distribute input data into hash tables. But with hash tables it is not important how much the output will change for input change as long as we receive uniformly filled tables for most of the input set.

Hash in Cryptology

Hash functions are very important in cryptography. But cryptography has some additional requirements for hash function which is hard to meet. Cryptographic hash functions can be divided into two larger parts. First part is called MDC. MDC means Manipulation Detection Code and as the name suggests it is used for detecting unauthorized manipulation with the message. The second part is called MAC, which means Message Authentication Code.

MAC was used to authenticate a message without any additional mechanism. MAC hash function has two inputs, which are data for hashing and the key. Therefore you can send a message and its hash. The receiver can compute hash of the message with the same key and if the hashes are equal the message originates from the owner of the key.

MDC is used in digital subscriptions algorithm. Asynchronous cryptographic algorithms are very slow and it would take too much time to encrypt the whole message which is unacceptable when subscribing a photograph, for example. Therefore the usual approach is to compute MDC hash from the message and subscribe only the hash. But to make this work the hash function has to meet three very important attributes.

- Preimage resistance
- 2-nd preimage resistance
- Collision resistance

Preimige resistance ensures that it is practically impossible to find any input which will result in a specific hash.

Hash function has 2-nd preimage resistance property if there is no practical possibility to compute another message, which has the same hash as the given message. Formally written it is $x \neq x_1 \wedge h(x) = h(x_1)$. If 2-nd preimage resistance and preimage resistance are true, then this hash function is called One way hash function.

Collision resistance property is true if there is no practical possibility of finding two messages which would have the same hash. This property is similar to 2-nd preimage resistance but for 2-nd preimage resistance the first message is given, but for collision resistance the attacker can choose both messages.

Practical impossibility is not a formal term because it is hard to formally define it. Practical impossibility means that the attacker will not able to find enough computation power to compute the result in a reasonable time. There are only few hash functions that have these properties. For more information about cryptographic hash function look at [23]

2.4.2 Cyclic Redundancy Check

CRC is used mainly for error detection. CRC stands for Cyclic Redundancy Code. For error detection using CRC the CRC hash is computed before sending the message and after receiving it. If both CRCs are equal then the transfer was performed without errors. This method requires transporting CRC with messages. Another method is modification of the message which rests in addition of CRC hash to the message. The message is moved to the left by the size of CRC and the CRC is used as the last bits of the message. In fact this is equal to transferring CRC with the message but this method computes CRC of the modified method and the result has to be equal to zero.

CRC is the remainder after polynomial division. More formal description of CRC can be found in [22]. For the detailed but informal description of CRC see [29].

Polynomial Arithmetic

Polynomial arithmetic is also called arithmetic in Galois Field. Carry is not defined in polynomial arithmetic. Let us have two numbers 14 and 19. Result of addition of these two numbers in classical arithmetic is 33 because $4 + 9$ is equal to 13 therefore the result in the first order is 3 and one is carried into the higher order. In polynomial arithmetic the result is 23 because there is no carry into the higher order. Carry is simply ignored.

CRC understands the input message as polynomial, which is divided by predefined divisor. All CRC calculations are done in GF(2) - Galois Field order 2. This means that the coefficient in polynomial can be only 0 or 1. In polynomial arithmetic in GF(2) addition and subtraction are the same and both can be implemented as logical function XOR. Thanks to this fact there is only partial ordering in this arithmetic.

```
A = 100101
B = 000101
C = 100011
```

It is easy to say that B¡C and B¡A, but C and A can not be compared by means of classical magnitude, because when adding or subtracting 000110 from A or C the result is the same, which is A in case that C was added and visa versa. Therefore, the only magnitude is magnitude of the highest bit. Division in polynomial arithmetic works as classical division. If the dividend is bigger than the divisor, the divisor is subtracted from the dividend until the remainder is less than the divisor.

This leads into the basic algorithm for CRC. Firstly, the message is written as a bit vector and then the generator polynomial, in other words the divisor, is put at the position of the first 1 in the message. Then the divisor is subtracted from the message simply by xorring each bit. Then the generator polynomial is moved so that its first 1 corresponds

with the first 1 in the message. This is repeated until the message is smaller than the generator polynomial. The remainder of such division is CRC.

This algorithm works fine, but has to compute XOR almost for each bit in the input message, sometimes even more than once. Therefore more efficient software algorithms were developed for CRC computing.

The size of the remainder is always the size of generator polynomial minus 1. Choosing the right generator polynomial can be hard. Most works recommend choosing one of the already published polynomials. But [17] shows that each polynomial has different behaviour in different tasks and therefore before choosing the right polynomial, a research should be done. Choosing a wrong polynomial can have catastrophic effect on performance.

2.4.3 Perfect hashing

Perfect hash function is a hash function which does not have any collisions in set S. Minimal perfect hash function is prefect hash function for which $n = m$. This means that minimal perfect hash function gives every key in the set a number in the range from 0 to the number of keys. Minimal perfect hash function has no hole in its returning interval. Therefore Minimal Perfect Hash Function (MPHF) would be ideal for the hash table. Unfortunately in many uses of hash table set S is not known and therefore it is not possible to find MPHF.

There are two groups of PHF or MPHF. One group is Order preserving while the other is not. Lets suppose that there is some order in set S. Then the output of an order preserving hash function will have defined the same order. This works also for partial order.

Theory of Perfect Hashing

This subsection briefly introduces the history of perfect hashing. Deeper survey of perfect hashing until 1992 can be found in [11].

It can be shown that probability that some function is PHF is approximately $\exp(\frac{n^2}{2m})$, where m is the number of position in PHF and n is the number of all keys. In fact this is another instance of a problem called "birthday paradox". This problem says that if there is group of 23 people, then there is 50% chance that two of them has birthday in the same day.

Probability of finding MPHF is $\exp(-n)$. It can also be used for computing how many bits must be stored to implement such MPHF. Therefore it is known that MPHF can be implemented with memory consumptions around 1.443 bits per key but no practical algorithm is known for construction of such function. Several theoretical algorithms are known but they work fine only for unreasonably high sizes of set S. Hagerup and Tholey [14] introduced an example of such algorithm. It can meet the border of 1.443 bits per key in its limit but it is not defined for n lesser than 2^{150}.

Perfect Hashing Based on Graph Theory

This approach was first introduced in [10]. There are many algorithms which use this approach but in this section only some of the most interesting are mentioned. Algorithms are called by the name of their authors because it is a common practice in their own publications.

In this approach to perfect hashing a normal hash function is usually generated. It would be hard work to generate a completely new hash function because such algorithm has to have a method to test if the newly generated function is really a hash function and

if this hash function is a good one. But there are several families of hash functions and algorithm can create only new representatives of a family of functions. [10] suggest using a table to generate hash functions. Tables of random integers are created for each hash function. These tables will have the same number of rows as the number of characters in the input alphabet. The table will have k columns, where k is length of the longest key from S. The result of hash function will be computed by this equation:

$$H_k(w) = (\sum_{i=1}^{|w|} T_k(i, w[i])) \mod m$$

If there is the need for creation of a new hash function, table T is filled with new random values only.

Let us have two normal hash function h_1 and h_2. Both of them are chosen randomly and the only requirements which they have to meet are fast execution time and uniform distribution of the input. h_1 and h_2 returns a number from the same interval M. These two hash functions define a random graph for set S. The random graph will have m vertices and n edges. Edges are defined by hash functions. For each element from set S, hash functions return two values. These values are the number of vertices which will be connected by a new edge. The second step of this algorithm requires the graph to be acyclic without loops and multiple edges. Therefore it has to be tested for these requirements. If the graph has these properties, then the algorithm continues with the second part. In other case new hash functions are generated together with a new graph. This is repeated until the created graph meets all our requirements. It can be shown from random graph theory that the probability of finding a graph with such property can be computed from the size of set S and interval M. Therefore the number of repetitions can be established by choosing the size of interval M. In practical implementations the size of M is chosen to be equal to cn, where $c \geq 2$. For $c = 2$ there will be probably two generations of hash function.

The second step comprises creating the minimal perfect hash function. Numbers can be added to each edge of the graph. This number results to MPHF and therefore has to be in the range from 0 to $n - 1$. The problem is how to store information about the edges. But it is easy to assign numbers to vertices of the graph. Then the number for each edge can be computed from the numbers stored in vertices of the graph. In implementation there is a table of size m. This table contains a number for each vertex. The number for the edge is computed as the sum of the numbers in vertices which belong to the edge. Due to the fact that the graph is acyclic, it is possible to compute the numbers of vertices in linear time. In fact this graph is a forest. It contains several subgraphs which are not connected. Therefore each subgraph can be done separately. Each such subgraph is a tree. This means that if the algorithm starts in the root vertex, it can go through the whole graph in such way that for every edge there is at least one vertex which is not evaluated. And therefore it can assign a number to this vertex which conforms to $h(w) = (g(h_1(w)) + g(h_2(w))) \mod n$ will be true. In this equation, function g represents a table which assigns a number to each vertex in the graph.

Second interesting algorithm using this approach is the algorithm called BMZ. This algorithm was introduced in [5]. Algorithm BMZ does not require an acyclic graph from the first step. This allows c to be chosen around 1. Smaller c leads to smaller memory consumptions. The use of a cyclic graph needs some changes in the second part of the algorithm. CHM is able to choose the output number for each key in set S. But this does not apply for BMZ. See figure 2.8 for explanation. If the first two edges in the figure are

solved, numbers are assigned to all vertices and therefore the number for the last edge is already assigned. It also means that minimal perfect hash function created by BMZ is not an order preserving hash function. The other side of the same problem is how to choose the right numbers for vertices of the graph. This can not be done in linear time for the cycle in the graph.

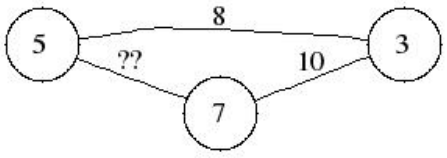


Figure 2.8: Value of the last edge in cycle is always set by values of previous edges

The graph can be divided into two parts. The first part contains cycles. Assigning numbers to the vertices of this part is more problematic and therefore it is called critical part of the graph. BMZ requires that the critical part of the graph is not bigger than the noncritical part of it. Naming process starts with the critical part of the graph. It can not be done in linear time. Naming is done by an iterative algorithm. In the first iteration all edges of the critical subgraph are named. Then the edges which have the same values are found and their vertices are reassigned. This step iterates until each two edges have assigned a different number. After this step approximately half of the edges have assigned their numbers. The other half of the numbers are scattered in the whole output interval. Assigning of the remaining edges is done in the same way as in CHM.

It is easy to show that in comparison with CHM, BMZ requires about half memory but still both of them require $n \log n$ memory. It is because both of them require to store m values in a range from 0 to $m-1$. Each value is stored on $\log m$ bits. Value m depends on the number of keys n. Therefore division of set S into k smaller sets from S_1 to S_k allows to obtain more optimal representation of MPHF. This idea is suggested in [7].

The last algorithm from this group is algorithm from [6]. This algorithm uses r hash function. Interval M is divided into r parts. Each hash function then returns numbers from its part of interval M. This means that first function returns numbers from interval $< 0, \frac{m}{r} - 1 >$ second function returns numbers from interval $< \frac{m}{r}, \frac{2m}{r} - 1$ and so on. Resulting hypergraph has to be acyclic and without multiple edges. The graph can not contain loops because in such case all hash functions have to return the same value which is impossible. Let us have function g which assigns a value from interval $< 0, r >$ to each vertex of the graph. Function g is created in creation of MPHF.

In evaluation of MHPF equation $p(x) = (\sum_{i=1}^{r} g(h_i(x)))) \mod r$ is computed. Function p gives a number of hash function which returns specific position of the input key in interval M. Because interval M is greater than n this is just PHF. To obtain minimal perfect hash function rank of $p(x)$ has to be computed. Function rank returns the number of previous valid p(x) results. By storing another m bits function rank can be computed in constant time. The result of function rank is MPHF.

The hardest step in creation of MPHF is finding function g. This step is referred to as the assigning step. Let V be boolean field size m which holds information if specified vertex was visited in the previous part of the assigning algorithm. V is initialized to false for each vertices. Let L be a list of all edges so that each edge in this list has at least one vertex

which is not in edges in the previous parts of the list. Then the algorithm goes through the whole list L from tail to head. Let u be an unvisited vertex belonging to the edge e and j is the position of vertex in the edge e. Then the value of function g for vertex u is computed by equation $g[u] = (j - \sum_{v \in e \wedge V[v]=true} g[v]) \mod r$.

Tests in [6] show that this algorithm has linear memory consumptions. For datasets of size about 3000000 keys this algorithm needs 2.6 bits per key for three hash functions and 3.6 bits per key for two hash functions. Memory consumption of this algorithm is given by choosing c. With the use of two hash function size of number c determines probability of finding an acyclic graph without multiple edges. But for three hash functions the probability of finding a hypergraph which meets these requirements changes by step function and therefore it is reasonable to choose c to the smallest number after this step. Therefore c is always chosen to 1.23.

Algorithm using Mapping, Ordering and Searching steps

The basic approach of this method is to divide the creation of MPHF into three steps which are called Mapping, Ordering and Searching. This approach was first presented in [12].

- Mapping transforms input of the hash function into reasonable intervals
- Ordering preprocess the result of the previous step and transforms it into a form which is most suitable for the last step
- Searching this step searches for MPHF function.

FCH algorithm [12] uses these three steps and obtains good results in the field of memory consumptions. FCH uses several normal hash functions. First of them h_{10} is used in Mapping step. This function assigns a number from range $< 0, n-1 >$ to each key from the input set. Then other two hash functions are used. Each of them accepts a number from this interval and returns the number of bucket, which will contain this key. The interesting part of this step is that the bucket is not filled uniformly. Almost 60% of all keys are mapped into 30% of all buckets while remaining 40% of keys are divided into 70% of buckets. This is done by two hash functions h_{11} and h_{12}. If the result of h_{10} is lesser than $0.6n$ the bucket number is returned by function h_{11}, which returns numbers only in interval $< 0, 0.3b - 1 >$, where b is the total number of buckets. In this is not the case, function h_{12} is used and it returns a number from $< 0.3b, b - 1 >$.

Ordering step in this algorithm only orders buckets by their sizes. In searching step it is not a problem to solve a big bucket if it comes first, while in the end of the searching step, small buckets are needed to fill holes in the output interval of MPHF.

Resulting MPHF is in form $h(k) = (h_20(k, d(B)) + g(B)) \mod n$, where B is the number of the bucket to which key was sorted during mapping step. Searching part of construction of the MPHF is looking for such functions g and d to ensure that there is no collision in the resulting interval. $d(B)$ returns only one bit which is used as a part of the seed in function h_{20} to avoid any collision in the bucket. Then function h_{20} creates some pattern which is rotated in the output interval until a position without collision is found. This is the reason why the algorithm has to start with big buckets. Searching step also choose seed which is most suitable for all buckets.

FCH algorithm has linear memory consumption and requires about 3.6 bits per key. Unfortunately its time complexity is not linear because the searching step is in fact searching in the whole state space and therefore has exponential time complexity in the worst case.

Practical Algorithms

This section contains description of several practical algorithms for perfect hashing. These algorithms have no theoretical background but were tested on specific problems and work fine. The problem is that there is no guarantee that their performance will be sufficient with different sets of keys.

Bloom Filters in Perfect Hashing This algorithm [21] implements several counting bloom filters. A counting bloom filter is an extension of a classical bloom filter. Each bit information in a bloom filter is extended with a counter which counts how many times this bit was set. There are several such bloom filters connected in serial fashion. When MPHF is created first the bloom filter has to learn all keywords from set S. There will be bits which were set only once by all keys. The key which sets such bit is removed from learning of the next bloom filters. Then the next bloom filter is taught to recognize the remaining set. This continues until the remaining set is empty.

A counter in the bloom filter is needed for creation of MPHF but not for its evaluation. Therefore only the bit field is stored for evaluation of MPHF. The respective bit in this field is set to 1 if the counter in the bloom filter contains 1 and to 0 if the counter contains any other value. During evaluation key is hashed into all bloom filters and the result of PHF is the position of first obtain 1. This is not MPHF because the bitfield probably contains many zeros, which is equal to a number which will never be returned. MPHF is obtained by counting all 1 which are predecessors of chosen positions. This is done by function rank and requires some additional time.

Perfect Hash in GPU Paper [19] presents the idea of multidimensional perfect hash function to compress textures in Graphic cards. Figure 2.9 shows the basic idea.

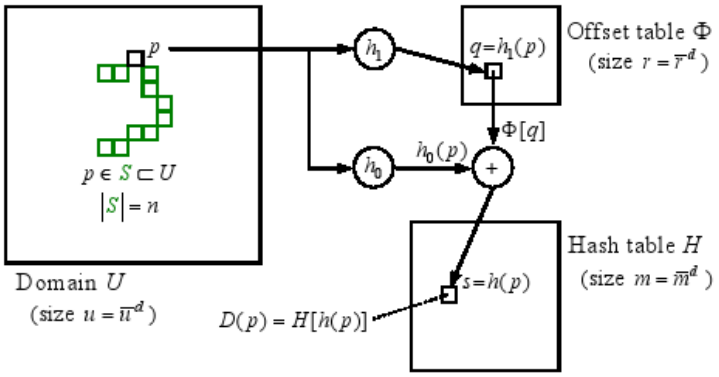


Figure 2.9: The idea of hashing in GPU from [19]

Function h_0 is a normal hash function which has collisions. Its collisions are solved by adding values from table r. Table r is addressed by another hash function h_1. Both hash functions are multidimensional and very easy to implement. In fact authors of the [19] discovered that implementing both hash functions as identities does not significantly hinder construction of a perfect hash function. Therefore both functions only wrap the spatial domain multiple times over offset and a hash table.

This algorithm has memory consumption in the interval from 3 bits per key up to 8 bits per key depending on the type of hashed data. Execution time of this algorithm is very fast.

Chapter 3

Proposed solution

3.1 Overview

Presented solution is based on Finite automatons because a Finite automaton can search through the input string in linear time complexity in the worst case. There are many methods which have better time complexity than Finite automatons but in the worst case they are the same or worse. Deterministic Finite Automatons were chosen because they allow changing the searched set simply by changing the transition table in memory. To obtain high throughput there is the need to accept several characters per cycle. Memory consumption of multichar automatons is too high and therefore the transition table has to be compressed. Snort rules often create many transitions which have the same start state and end state. Such transitions differ only by characters which they accept. All these transitions can be represented by one transition which has assigned a set of all characters. This combined with a shared decoder significantly saves memory. The problem with shared decoders is that the decoder has to be able to choose one set for each input character. The basic concept of shared decoder is shown in figure 3.1. Shared decoder works without knowledge of actual state in automaton and could be shared between several automatons.

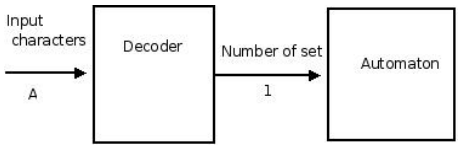


Figure 3.1: Architecture of a searching unit with a shared decoder

Figure 3.2 is a simple example of classical NFA. This Automaton requires 8 records in the transition table. The transition table could be reduced into two records by using previously mentioned compression. The result can be seen in figure 3.3. Such Automaton can not be used with shared decoder because the decoder is not able to deterministically assign A to any of possible set. Therefore alphabet of automaton must be change to Deterministic alphabet. The final result can be seen in figure 3.4.

Creation of Minimal DFA with a shared decoder which accepts several characters per cycle is done in several steps and is shown in figure 3.5.

Snort has its rules defined as regular expressions. Translations of regular expressions into a Finite Automaton is done by a parser made by Andrej Hank [15]. Therefore this

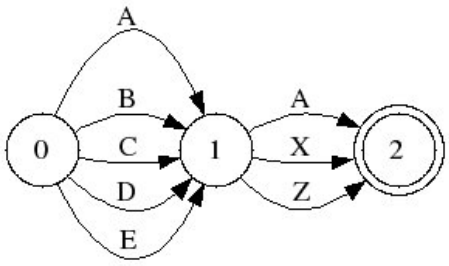


Figure 3.2: Classical NFA

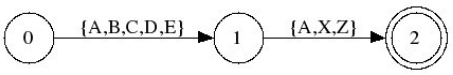


Figure 3.3: NFA with a compressed transition table

work starts with Nondeterministic Finite Automaton for each rule. Because it is impossible to have a processing unit for each rule, a method for finding suitable pattern sets was implemented. Unfortunately this method requires computing Minimal Deterministic Finite Automaton with Deterministic alphabet for each possible couple of regular expressions. If the number of characters accepted in each cycle varies then each couple has to be recomputed. This means that it is not possible to find suitable sets of regular expressions with automatons accepting only one character per cycle and then use it for ten characters per cycle.

3.2 Construction of Minimal Deterministic Final Automaton from ruleset

This is the fundamental operation of the whole approach. The input for this operation is a set of Nondeterministic Finite Automatons. The basic idea of construction is implemented as shown in figure 3.5. The only missing step is joining all patterns from the step into one big NFA which is used as the input to the algorithms. Joining is realized by function Together. This function expects its input in the form of two Nondeterministic Automatons in the format developed by Andrej Hank in his parser. Function Together returns join of these two automatons through its third parameter. The output automaton is in the same format as the inputs. This function is repeatedly used until there is an automaton which is not joined into the result.

The second step is creation of a table representation of NFA. This is done by function nfa2TableNFA. This function takes an automaton as its parameter and returns a pointer to the table representation of NFA. The table representation uses standard library types such as vector and map to implement the transition table. All implemented functions which work with NFA use this representation.

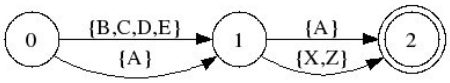


Figure 3.4: NFA with a compressed transition table with deterministic alphabet

NFA has to be converted into its epsilon free representation by calling function efreeNFA. This function takes the pointer to NFA as its only parameter and returns the pointer to a newly created NFA in the same representation but without epsilon transitions.

Now it is possible to convert the automaton into Multichar. This is done by procedure Multichar which requires the table representation of NFA and the number of characters which should be accepted by one transition. Function Multichar returns the pointer to the table representation of the multichar automaton.

The result of Multichar is used by function DeterLong. This function ensures that the alphabet is Deterministic. Formal definition of Deterministic alphabet is in 3.5. Function DeterLong changes the alphabet of the input NFA to be deterministic and therefore it has to add transition into resulting NFA. This function returns the pointer to a newly created table representation of NFA.

The last two functions are DFA and MinDFA. These functions determinize and minimize NFA respectively DFA. DFA is represented by its own type.

3.3 Dividing Rules into Several DFA

The basic idea of this approach was described in theoretical background in 2.3.2. The problem with the method introduced by [30] is that their heuristics works only with one bit information. If the join of two automatons is greater than the sum of their sizes then an edge is added into the graph, otherwise not. This work extends their approach. An edge was created between all states of the graph and these edges were labeled. Each edge has a floating point number which is computed as $\frac{sizeofjoin}{sumofsizes}$. When choosing which rule has to be added into the group, labels of the edges are multiplied and the rule is chosen. It has to have the minimal number of all other rules in case of the first rule in the group or it has to have minimal connection with rules in group.

It is also possible to label edge by absolute number. In such case it is the difference between the number of states of the joined automaton and the sum of states in both original small automatons.

The experiment shows that the first approach results in more compact sets and therefore it is implemented. Dividing into groups is implemented by function Groups which takes the vector of all small automatons as its first parameter. Second parameter of this function is the maximal size of Minimal DFA representing set and the third parameter holds the number of characters accepted by one transition.

3.4 Multichar Finite Automaton

Creation of a Multichar Finite Automaton is realized by function Multichar. This function takes a table representation of classical NFA and turns it into a Multichar. The first step of this process is to compute transitions of the new automata. This is done by recursive

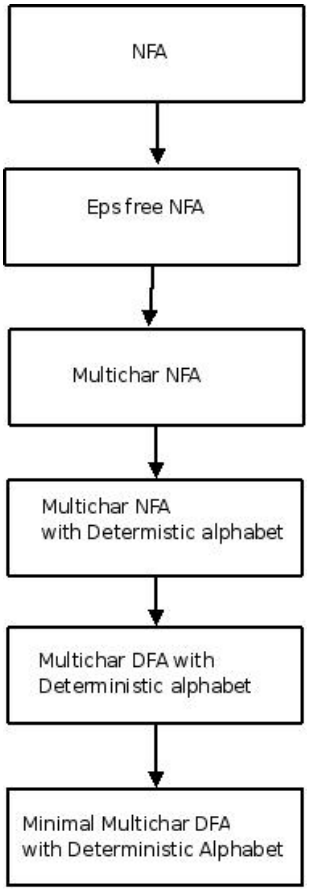


Figure 3.5: Algorithm of creating Minimal DFA with a shared decoder which accepts multiple characters per cycle

functions and the recursion depth is defined by the number of characters accepted by one transition. A description of this algorithm is in theoretical section 2.3.1.

Function Multichar has to change the alphabet of the automaton too. The alphabet of the automaton is stored in a vector of characters which is a part of the table representation of NFA. The multichar has to change it to a vector of vectors of characters because after multichar each symbol is a vector of characters. For easy implementation data type of the alphabet remains the same for automaton receiving one character per transition and multiple characters per transition. They differ only in length of the vector which represents the symbol.

Implementation of function Multichar uses types from STL such as vector and set. This allows fast developing and changing the function but also limits the speed of Multichar execution.

3.5 Deterministic Alphabet of Finite Automaton

As mentioned above it is crucial to have a method for compression of the transition table. The basic method proposed in this work is the representation of transition symbol by set of characters. By this compression it is possible to save the number of bits which equals to

$2\log(m)$ where m is the number of states per each character added into the set.

It was found that many of the sets proposed in the previous paragraph are the same for Snort rules. Therefore the idea of a shared decoder was developed to bring further additional memory savings. Basically, the shared decoder translates the alphabet of the string into a new alphabet which is used by the automaton. The automaton with a shared decoder uses numbers as its alphabet. Therefore the number of bits required for the transition table is equal to $n(2\log(m)+\log(k))$ where n is the number of transitions, m is the number of states in the automaton and k is size of the alphabet. It can be seen that the biggest memory savings are due to the fact that the whole set is represented by one number.

The shared decoder has an input string at its input and translates it into a succession of numbers which represents a symbol in the alphabet of the automaton. This translation has to be unambiguous. This means that for each possible input character the shared decoder has to return one number. As it is clear from figure 3.3 such mapping is not always possible. Alphabet which allows construction of a shared decoder is called Deterministic alphabet M which is defined as

$$\forall x \in M \wedge \forall y \in M : \text{ if } x \neq y, \text{ then } x \cap y = \emptyset$$

It is always possible to transform nondeterministic alphabet of an automaton to its deterministic variant. Deterministic alphabet is bigger than nondeterministic which leads to a bigger transition table of the automaton.

3.6 Multichar Deterministic Alpabet of Finite Automaton

In a multichar automaton a symbol is defined as succession of characters. After introducing the compression method from previous character Symbol of the multichar automaton must be defined as a succession of sets of characters. When A is Symbol then A_i refers to i-th set of Symbol A.

Length of Symbol A is then defined as $legth(A)$ number of set in Symbol.

Size of Symbol A is defined as $|A| = \sum |A_i|$

Symbols A and B are equal when

$$(A = B) \Leftrightarrow (length(A) = length(B) \wedge \forall i < length(A) : A_i = B_i)$$

Intersection of symbols A and B is defined as

$$(P = A \cap B) \Leftrightarrow ((lenght(A) = length(B)) \wedge \forall i < length(A) : P_i = A_i \cap B_i \wedge P_i \neq \emptyset)$$

The last operation defined with symbols is subtraction. Let A and B be symbols of the same length and let x be succession of 0 or 1 of the same length. Then X is the set of all possible x. Subtraction of symbol is defined only if intersection of symbols is defined. Let $P = A \cap B$. Then $A - B = M$, where M is set such as

$$S_1 = \{\text{Symbol } N : \forall i < length(A) \wedge \forall x \in X : \text{ if } x = 1 \text{ than } N_i = A_i/P_i \text{ else } N_i = P_i\}$$
$$S_2 = \{\text{Symbol } N : \forall i < length(A) \wedge \forall x \in X : \text{ if } x = 1 \text{ than } N_i = B_i/P_i \text{ else } N_i = P_i\}$$
$$M = S_1 \cup S_2$$

Deterministic alphabet can be defined in a similar way as in the previous paragraph. Differences in definition are caused by different definition of intersection. Intersection for

Symbols is never empty and therefore the definition only tests existence of intersection. Alphabet M is deterministic if and only if

$$\forall x \in M \wedge \forall y \in M : \text{ if } x \neq y \text{ then do not exist } P \text{ such as } P = A \cap B$$

Transformation from a normal alphabet M_{in} into deterministic alphabet M_{out} is done by this simple algorithm:

1. Create queue F_1 which contains all Symbols from M_{in} ordered by size
2. Create empty alphabet M_{out}
3. While F_1 is not empty do
   - Remove first Symbol from F_1 and call it A
   - Call function AddSym(A,M_{out})

Function AddSym adds symbol into the deterministic alphabet. Each empty alphabet is deterministic but by adding new symbols deterministic property can be lost. Therefore AddSym has to test if after adding a new symbol the alphabet still remains deterministic. If not, the symbol is divided into several smaller symbols and each of them is added independently.

Function AddSym:

1. Create set k such that $k = \{x$, where $x \in M \wedge \exists P$, such as $P = x \cap A\}$
2. If $k = \emptyset$ than $M_{out} = M_{out} \cap \{A\}$
3. If $k \neq \emptyset$ than $n \in k$
   - If $n = A$ then end function
   - If $n \neq A$ then
     - remove n from M_{out}
     - $NSym = n - A$
     - $\forall y \in NSym$: call function AddSym(y,M_{out})

3.7 Perfect hashing and transition table

Even with proposed optimizations transition tables are too big. Even for an automat accepting 4 characters per transition build from 10 Snort rules size of the transition table exceeds 20000 records. With limited memory in FPGA it is not possible to store the whole transition table for an arbitrary automaton, not to mention that for 10Gb/s networks, created automaton has to accept more than 14 characters per transition. Therefore FPGA which implements searching unit has to be equipped with external memory. Access to external memory is slow and therefore it is not possible to have several accesses to external memory just to determine the next state.

One of possible solutions can be perfect hashing. Next State logic in FPGA implements only Minimal Perfect Hash function. The result of MPHF points to external memory, where the transition is stored. In such case only one access to external memory is needed to obtain the next state. If the external memory does not contain data then it means that

such transition is not in the transition table. In other cases external memory contains the next state.

In the theoretical background several algorithms for perfect hashing were introduced. BMZ or CHM algorithm can not be used because of their memory consumption. Both of them have memory complexity $n \log(n)$, where n is the number of records in the transition table. This means that for 20000 records, memory needed for MPHF would be more than 2Mb. This can be solved by could be dividing MPHF into many smaller MPHF or use another algorithm. Another theoretically well understood algorithm was FCH and the algorithm introduced [6]. These two algorithms have linear memory consumption and tests in [6] show that these algorithms enable construction of MPHF which needs only about 3 bits per record. The disadvantage of [6] is the need for computation of function rank to obtain MPHF from PHF. This requires some time in evaluation and also requires m bits in MPHF. This disadvantage could be solved by implementing only PHF but then we would need about 1.23 times more external memory.

There are several other algorithms for creating MPHF introduced in 2.4.3. The disadvantage of these algorithms is the impossibility of finding a theoretical proof that their memory consumption remains low for different data sets.

Chapter 4

Experimental results

Automaton for single rule with different number of characters per transition

This experiment shows how is transition table increased by increasing number of characters accepted by one transition. Experiment was done for randomly selected rule from Snort database.

```
/(ntscan [0-9]{1,4} [0-9]{1,4}|dcom\.self|scan\.(start|stop)|scan ([0-9]{1,3}\.
[0-9]{1,3})|(advscan|asc|xscan|xploit|adv\.start) (webdav|netbios|ntpass|dcom
(2|135|445|1025)|mssql|lsass|optix|upnp|dcass|beagle[12]|mydoom|netdevil|
DameWare|kuang2|sub7|iis5ssl|wkssvc|wks1|mysql|wkssvcOth|wkssvcENG|arkeia|
arcserve|wins|veritas|netbackup|asn1))/i
```

Characters per transition	Transitions in NFA	Transitions in NFA with deterministic alphabet	Transitions in DFA
2	549	1667	841
3	708	5852	2839
4	872	32711	13074
5	1173	141087	51890
6	1363	672715	234468

First column of the table contains number of characters which is accepted by each transition. Second column contains number of transition in classical NFA. Third column contains number of transition on NFA after transforming alphabet into its deterministic form. Last column contains number of transition in DFA created from NFA with deterministic alphabet.

It could be sen from the table that deterministic alphabet has much more symbols than nondeterministic alphabet. Deterministic alphabet is crucial for using shared decoder and using shared decoder brings great memory savings for transition table because without shared decoder each state has to store its own set of symbols. Number transition has to be multiply by size of the symbol of the transition. This experiment shows the need for very efficient searching in transition table.

Dividing patterns into sets

In this experiment it is shown how dividing into sets works for the snort patterns.

Character per transition	Limit for set	Number of sets	Sum of transitions in all sets	Sum of Symbols in all alphabets
2	6000	5	17461	2288
3	6000	10	33857	8621
3	16000	6	50833	10770

First column of this table contains number of character accepted by each transition. Second column contains limit for the size of each set. Third column contains number of created sets. Fourth column contains Sum of all transitions in all sets and last column is sum of all symbols in all alphabets.

This experiment shows that lesser number of the sets means more transition and more complex alphabet. This is because rules are joined together until result is smaller than specified limit. It is possible that result of the merge of two automaton will collide with some rule which does not collide with any of the small automatons. Another reason of this phenomenon is that smaller number of the sets creates more complex automatons for each set.

In this implementation number of set is chose by algorithm. In real implementation it would be static and therefore it is better that algorithm creates little more complex automaton than many numbers of the sets.

Comparison of heuristic for dividing patterns into sets

There is several possible heuristic described in previous part of this work. Each of this heuristics was used for dividing set of snort rules. Quality of the heuristic is mesured by number of set which is produced by algorithm.

Character per transition	Limit for set	Number of sets by original heuristics	Number of sets by absolute heuristic	Number of sets by standard heuristic
2	6000	8	6	5
3	6000	13	9	10

First two columns have the same meanings as in previous experiments. Third column contains number of sets created by heuristics introduced in [30]. Fourth and fifth column contain results obtain by modifications presented in this work. Fourth column uses absolute difference in number of state of automaton created by joining two rules and sum of number of states of these automatons. While heuristics in fifth column normalize difference by size of these automatons.

This experiment shows that both presented modifications has better results than original heuristics. While difference between two modification is insignificant. Quality of the results depends on the dividing set therefore both heuristics can be used. Unfortunately it is not possible to use both heuristics and then choose better result because of computation complexity.

Chapter 5

Conclusion

Main purpose of this work was to experimentally test the possibility of implementing Deterministic Finite Automatons for searching in the payload of modern networks.

As a part of this work the library for Finite Automatons was implemented in C++. This library was not optimized for speed and its optimization could be a task for a possible future work. The experiments indicate that the most time expensive operation is creation of deterministic alphabet for the finite automaton. This function repeatedly calls intersection of set of char. An optimal implementation of this operation could bring extremely high acceleration of this function. Another possible way for optimizing developed library is to implement the table representation of NFA without STL data types. STL types such as set or vector have extremely high memory consumption and therefore the work with big multichar automatons is impossible. By optimizing memory consumption of the table representation it could be possible to load bigger part of an automaton into cache memory of processor which would bring another additional acceleration.

A large part of this work is also dedicated to perfect hashing. It is shown that perfect hashing could be used for implementing next state logic in DFA. Several algorithms for perfect hashing which can be used for transition table indexing were introduced in the theoretical section.

Unfortunately, even with introduced optimizations, the resulting DFA are still too big for implementation in FPGA without additional memory. Therefore future work could also search for other possible optimizations of DFA.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search, 1975.

[2] M. Attig, S. Dharmapurikar, and J. Lockwood. Implementation results of bloom filters for string matching, 2004.

[3] Z. K. Baker and V. K. Prasanna. Automatic synthesis of efficient intrusion detection systems on fpgas, 2004.

[4] Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on fpgas, 2004.

[5] Fabiano C. Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. A practical minimal perfect hashing method, 2005.

[6] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions.

[7] F.C. Botelho, D. Belazzougui, and Davi de Castro Reis. External memory based algorithm, http://cmph.sourceforge.net/brz.html.

[8] Thierry Lecroq Christian Charras. Exact string matching algorithms.

[9] Ch. Clark and D. Schimmel. Scalable pattern matching for high-speed networks, 2004.

[10] Z.J. Czech, G. Hawas, and B.S. Majewski. An optimal algorithm for generation minimal perfect hash functions, 1992.

[11] Z.J. Czech, G. Hawas, and B.S. Majewski. Fundamental study perfect hashing, 1997.

[12] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions, 1992.

[13] M Franc. Webový formulář pro minimalizaci nedeterministických konečných automatů.

[14] T. Hagerup and T.Tholey. Efficient minimal perfect hashing in nearly minimal space, 2001.

[15] Andrej Hank. Detekcia narušenia počítačovej siete, bakalářská práce, 2007.

[16] P Kobiersky. Návrh architektury ids systému pro technologii fpga.

[17] Philip Koopman and Tridib Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks.

[18] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection, 2006.

[19] S. Lefebvre and H. Hoppe. Perfect spatial hashing, 2006.

[20] Ted G. Lewis and Curtis R. Cook. Hashing for dynamic and static internal tables.

[21] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications.

[22] Petr Matoušek. Cyklické kódy - kódování a dekódování.

[23] Vanstone Menezes, Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, 1997.

[24] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Lecture Notes in Computer Science*, 2161, 2001.

[25] J. Strother Moore Robert S. Boyer. A fast string searching algorithm, 1977.

[26] Snort. Snort project www page. http://www.snort.org.

[27] Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection.

[28] J Tobola. Vyhledávání řetězců v payloadu paketu s využitím tcam.

[29] Ross N. Williams. A painless guide to crc error detection algorithms.

[30] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection.

[31] M Čéška. Teoretická informatika.